

Embedding by Normalisation

Shayan Najd

LFCS,
The University of Edinburgh
sh.najd@ed.ac.uk

Sam Lindley

LFCS,
The University of Edinburgh
sam.lindley@ed.ac.uk

Josef Svenningsson

Functional Programming Group,
Chalmers University of Technology
josefs@chalmers.se

Philip Wadler

LFCS,
The University of Edinburgh
wadler@inf.ed.ac.uk

Abstract

This paper presents the insight that practical embedding techniques, commonly used for implementing Domain-Specific Languages, correspond to theoretical Normalisation-By-Evaluation (NBE) techniques, commonly used for deriving canonical form of terms with respect to an equational theory.

NBE constitutes of four components: a syntactic domain, a semantic domain, and a pair of translations between the two. Embedding also often constitutes of four components: an object language, a host language, encoding of object terms in the host, and extraction of object code from the host.

The correspondence is deep in that all four components in embedding and NBE correspond to each other. Based on this correspondence, this paper introduces Embedding-By-Normalisation (EBN) as a principled approach to study and structure embedding.

The correspondence is useful in that solutions from NBE can be borrowed to solve problems in embedding. In particular, based on NBE techniques, such as Type-Directed Partial Evaluation, this paper presents a solution to the problem of extracting object code from embedded programs involving sum types, such as conditional expressions, and primitives, such as literals and operations on them.

Categories and Subject Descriptors D.1.1 [Applicative (Functional) Programming]; D.3.1 [Formal Definitions and Theory]; D.3.2 [Language Classifications]: Applicative (functional) languages

Keywords domain-specific language, DSL, embedded domain-specific language, EDSL, semantic, normalisation-By-evaluation, NBE, type-directed partial evaluation, TDPE

1. Introduction

Less is more sometimes. Compared to General-Purpose Languages (GPLs), Domain-Specific Languages (DSLs) are smaller and simpler. Unlike GPLs, DSLs are designed ground up to describe programs used in a specific domain. DSLs are a powerful engineering tool: DSLs abstract over domain-specific concepts and operations by providing a set of primitives in the language.

Embedding a DSL in a host GPL is by now well established as a family of techniques for simplifying its implementation. Embedded DSLs can reuse some of the existing machinery implemented for their host language; for a particular Embedded DSL (EDSL),

one does not need to implement all the required machinery. For instance, by using quotations (Najd *et al.* 2016; Mainland 2007), macros (Flatt 2010), overloading and virtualisation of constructs (Wadler and Blott 1989; Rompf *et al.* 2013), or normal techniques for modular programming (Rossberg 2015), there is no need for implementing a parser; by using higher-order abstract syntax and piggybacking on module system of host, there is no need to implement a name-resolver; and, by using mechanisms such as Generalised Algebraic Data Types (GADTs) (Cheney and Hinze 2003), there is no need to implement a type-checker.

Furthermore, as with any other host program, EDSL programs can often integrate smoothly with other host programs, and reuse parts of the ecosystem of the host language. Language-INtegrated Query (LINQ) (Meijer *et al.* 2006b) is a well-known instance of such successful integration, where SQL queries, as embedded DSL programs, are integrated with programs in mainstream GPLs.

Although embedding can avoid remarkable implementation effort by reusing the machinery available for the host language, it comes with the expensive price of EDSLs losing their authentic identity. Compared to stand-alone implementation, embedding is less flexible in defining syntax and semantic of DSLs; more or less, syntax and semantic of EDSLs often follow the ones of the host language. There are variety of smart and useful techniques to partially liberate EDSLs from such restrictions (e.g., see Najd *et al.* (2016); Reynolds (1998); Svenningsson and Axelsson (2015); Axelsson (2012); Rompf and Odersky (2010)). However, employing clever embedding techniques and stacks of unconventional features available in the host language, can lead to cryptic code sometimes. It becomes difficult to distinguish an EDSL from the the host language, as the boundary between an EDSL and its host language would not be entirely clear. Implementation-based descriptions make EDSL rather difficult to learn, not only for domain experts, whom traditionally are assumed to be unfamiliar with the host language, but also for the host language experts unfamiliar with the domain. Whole is nothing without all its parts, and whole is greater than the sum its parts.

Unlike stand-alone languages that are often accompanied by a set of formal descriptions, EDSLs are often presented by actual code in a mainstream host language. Also, the embedding techniques themselves are described in terms of a unique set of language features they employ. For instance in Haskell, a deep embedding means datatypes in the host language are used for representing the syntax of EDSLs, and functions (programs in general) over the datatypes are used for defining semantics; or, in a final tag-less embedding (Carette *et al.* 2009) type-classes are used to define an interface representing syntax, and instances of the type-classes are used for defining semantics.

Implementation-based descriptions make embedding techniques rather difficult to learn as well: techniques vary greatly from one host language to another, and even in a host language it is difficult to compare techniques. As a result, existing techniques are hard to

scale. For instance, once one moves from embedding simpler DSLs to DSLs with richer computational content, it becomes harder for embedding to stay close to the intended syntax and semantic on one hand, and reuse the host machinery on the other. Would it not be convenient to have a more formal and implementation-independent description of EDSLs and embedding techniques? This paper is taking a few steps toward this goal.

The kind of principles and descriptions this paper is aiming for is the ones of mathematical nature: abstract, insightful, and simple. These are the kind of principles that have been guiding design of functional programs since their dawn. One may argue these principles are the ones that are discovered, as opposed to being invented (Wadler 2015).

For instance, Carette *et al.* (2009, p. 513) observes that final tagless embeddings are semantic algebras and form fold-like structures. This observation has been explored further by Gibbons and Wu (2014), where they identify shallow embedding as algebras of folds over syntax datatypes in deep embedding. Decomposing embedding techniques into well-known structures such as semantic algebras or folds is liberating: embedding techniques can be studied independent of language features. Semantic algebras and folds enjoy clear mathematical and formal descriptions (e.g., via categorical semantics), hence establishing correspondence between embedding and folds enables borrowing ideas from other related fields.

In pursuit of a formal and implementation-independent description of practical embedding techniques, this paper proposes Embedding-By-Normalisation, EBN for short, as a principled approach to study and structure embedding. EBN is based on a direct correspondence between embedding techniques in practice and Normalisation-By-Evaluation (Martin Löf 1973; Berger and Schwichtenberg 1991) (NBE) techniques in theory.

NBE is a well-studied approach (e.g., see Altenkirch *et al.* (1995, 2001); Dybjer and Kuperberg (2012); Lindley (2005)) in proof theory and programming semantics, commonly used for deriving canonical form of terms with respect to an equational theory. Decomposing embedding techniques into the key structures in NBE is liberating: embedding techniques can be studied independent of language features and implementations. NBE enjoys clear mathematical and formal description, and the connection between NBE and EDSLs allows for transferring results from one field to the other, thereby strengthening both. For instance, in this paper we show how to use the NBE technique Type-Directed Partial Evaluation (TDPE) (Danvy 1996b) to extract object code from host terms involving sums types, such as conditional expressions, and primitives, such as literals and operations on them. Although, there may exist various smart practical solutions to the mentioned code extraction problem; at the time of writing this paper, the process of code extraction for sum types and primitives is considered an open theoretical problem in the EDSL community (see Gill (2014)).

The contributions of this paper are as follows:

- To characterise the correspondence between Normalisation-By-Evaluation (NBE) and embedding techniques (Section 2 and 3)
- To introduce Embedding-By-Normalisation (EBN) as a principled approach to study and structure embedding inspired by the correspondence to NBE (Section 2 and 3)
- To propose a simple parametric model capturing a large and popular class of EDSLs, and introducing a series of EBN techniques for this model (Section 4)
- To show how to extract code from embedded terms involving sum types, such as conditional expressions, as a by-product of EBN for above model involving sum types (Section 4.2)

- To show how to extract code from embedded terms involving primitive values and operations, as a by-product of EBN for above model involving primitives (Section 4.3)
- To show how EBN relates to some of the related existing techniques, and highlighting some insights when observing such techniques through EBN lens (Section 5)

To stay formal and implementation-independent, the descriptions and code in the main body of this paper is presented using type theory, following a syntax similar to the one in the proof assistant Agda (Norell 2009). Only a minimal set of language features is used, hoping for the presentation to remain accessible to the readers familiar with functional programming. When inferable from context, some unnecessary implementation details, such as type instantiations or overloading of constants, are intentionally left out of the code for brevity. The implementation concerns are addressed separately throughout the paper. Code and definitions presented in this paper are implemented in Agda, and are available at <https://github.com/shayan-najd/Embedding-By-Normalisation>.

2. Normalisation-By-Evaluation

Normalisation-by-Evaluation (NBE) is the process of deriving canonical form of terms with respect to an equational theory. The process of deriving canonical forms is often referred to as normalisation, where the canonical forms are referred to as normal forms. NBE dates back to Martin Löf (1973), where he used a similar technique, although not by its current name, for normalising terms in type theory. Berger and Schwichtenberg (1991) introduced NBE as an efficient normalisation technique. In the context of proof theory, they observed that the round trip of first evaluating terms, and then applying an inverse of the evaluation function, normalises the terms. Following Berger and Schwichtenberg (1991), Danvy (1996b) used NBE to implement an offline partial evaluator that only required types of terms to partially evaluate them.

There are different approaches to normalisation. One popular approach to normalisation is reduction-based, where a set of rewrite rules are applied exhaustively until they can no longer be applied. In contrast to reduction-based approaches, NBE is defined based on a pair of well-known program transformations, instead of rewrite rules. For this reason, NBE is categorised as a reduction-free normalisation process.

NBE constitutes of four components:

Syntactic Domain is the language of terms to be normalised by a NBE algorithm.

Semantic Domain is another language used in NBE, defining a model for evaluating terms in the syntactic domain. Often the semantic domain contains parts of the syntactic domain left uninterpreted. The uninterpreted parts are referred to as the residual parts, and in their presence the semantic model as the residualising model.

Evaluation is the process of mapping terms in the syntactic domain to the corresponding elements in the semantic domain. Despite the name, the evaluation process in NBE is often quite different from the one in the standard evaluators. Although it is not necessarily required, the evaluation process in NBE is often compositional. In this paper, following the convention, evaluation functions are denoted as $\llbracket _ \rrbracket$. In the typed variant of NBE, same notation is also used to denote mapping of types in evaluation.

Reification is the process of mapping (back) elements of semantic domain to the corresponding terms in the syntactic domain. In this paper, following the convention, reification functions are denoted as \downarrow .

More formally, an algorithm with NBE structure can be seen as an instance of the following (dependent) record:

$$\text{NBE} = \{ \text{Syn} : \text{Type}, \\ \text{Sem} : \text{Type}, \\ \llbracket _ \rrbracket : \text{Syn} \rightarrow \text{Sem}, \\ \downarrow : \text{Sem} \rightarrow \text{Syn} \}$$

As mentioned, normalisation in NBE is the round trip of first evaluating terms, and then reifying them back. Therefore, normalisation in NBE is a mapping from syntactic domain to syntactic domain:

$$\text{norm} : \text{Syn} \rightarrow \text{Syn} \\ \text{norm } M = \downarrow \llbracket M \rrbracket$$

In a typed setting, it is expected that transformations in NBE to preserve types of the terms. More formally, an algorithm with NBE structure in a typed setting can be seen as an instance of the following (dependent) record, with the following normalisation function:

$$\text{TypedNBE} = \{ \text{Syn} : \underline{\text{Type}} \rightarrow \text{Type}, \\ \text{Sem} : \underline{\text{Type}} \rightarrow \text{Type}, \\ \llbracket _ \rrbracket : \forall A. \text{Syn } A \rightarrow \text{Sem } A, \\ \downarrow : \forall A. \text{Sem } A \rightarrow \text{Syn } A \} \\ \text{norm} : \forall A. \text{Syn } A \rightarrow \text{Syn } A \\ \text{norm } M = \downarrow \llbracket M \rrbracket$$

Above, Type denotes kind of object types. It is underlined to contrast it with Type which is the kind of types in the metalanguage.

A valid NBE normalisation algorithm, both untyped and typed, should guarantee that, (a) normalisation preserves the intended meaning of the terms, and (b) normalisation derives canonical form of terms up to certain congruence relation.

2.1 A First Example

As the first example, consider the "hello world" of NBE, terms of the following language:

$$c \in \text{Char} \text{ (set of characters)} \\ L, M, N \in \text{Chars} ::= \epsilon_0 \mid \text{Chr } c \mid M \bullet N$$

The language, referred to as Chars, consists of an empty string, a string containing only one character, and concatenation of strings. For example, the terms

$$\text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet \text{Chr 'E'})$$

and

$$(\text{Chr 'N'} \bullet \epsilon_0) \bullet ((\text{Chr 'B'} \bullet \epsilon_0) \bullet (\text{Chr 'E'} \bullet \epsilon_0))$$

both represent the string "NBE".

The intended equational theory for this language is the one of free monoids, i.e., congruence over the following equations:

$$\begin{aligned} \epsilon_0 \bullet M &= M \\ M \bullet \epsilon_0 &= M \\ (L \bullet M) \bullet N &= L \bullet (M \bullet N) \end{aligned}$$

NBE provides a normalisation process to derive a Canonical form for the terms with respect to above equational theory. If two terms represent the same string, they have an identical canonical form. For instance, the two example terms above normalise to the following term in canonical form:

$$\text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet (\text{Chr 'E'} \bullet \epsilon_0))$$

For a specific syntactic domain, in this case the Chars language, there are different ways to implement a NBE algorithm, as there

are different semantic domains to choose from. For pedagogical purposes, two distinct NBE algorithms are presented for the Chars language based on two distinct semantic domains: (1) lists of characters, and (2) functions over the syntactic domain itself.

2.1.1 Lists as Semantic

The syntactic domain given to be the Chars language, and semantic domain chosen to be a list of characters, the next step for defining a NBE algorithm is defining an evaluation function:

$$\begin{aligned} \llbracket _ \rrbracket : \text{Chars} &\rightarrow \text{List Char} \\ \llbracket \epsilon_0 \rrbracket &= [] \\ \llbracket \text{Chr } n \rrbracket &= [n] \\ \llbracket M \bullet N \rrbracket &= \llbracket M \rrbracket ++ \llbracket N \rrbracket \end{aligned}$$

Evaluation defined above is a simple mapping from Chars terms to lists, where empty string is mapped to empty list, singleton string to singleton list, and concatenation of strings to concatenation of lists. For instance, the two example terms representing "NBE" earlier are evaluated to the list ['N', 'B', 'E'].

Above evaluation process is particularly interesting in that it is compositional: semantic of a term is constructed from the semantic of its subterms. Though compositionality is a highly desired property, thanks to the elegant mathematical properties, the evaluation process in NBE is not required to be compositional. In fact, some evaluation functions cannot be defined compositionally. Compositionality of a function forces it to be expressible solely by folds, and not every function can be defined solely in terms of folds. For instance, evaluation that rely on some forms of global transformations, sometimes cannot be expressed solely in terms of folds.

The next step is to define a reification process:

$$\begin{aligned} \downarrow : \text{List Char} &\rightarrow \text{Chars} \\ \downarrow [] &= \epsilon_0 \\ \downarrow (c :: cs) &= \text{Chr } c \bullet (\downarrow cs) \end{aligned}$$

Reification defined above is a simple mapping from lists to Chars terms, where empty list is mapped to empty strings, cons of list head to list tail to concatenation of the corresponding singleton string to the reified string of tail. For example, the list ['N', 'B', 'E'] from earlier is reified to the following term:

$$\text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet (\text{Chr 'E'} \bullet \epsilon_0))$$

The reification function is also compositional.

Putting the pieces together normalisation function is defined as usual:

$$\begin{aligned} \text{norm} : \text{Chars} &\rightarrow \text{Chars} \\ \text{norm } M &= \downarrow \llbracket M \rrbracket \end{aligned}$$

As expected, above function derives canonical form of Chars terms. For instance, we have

$$\begin{aligned} \text{norm } (\text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet \text{Chr 'E'})) &= \\ \text{norm } ((\text{Chr 'N'} \bullet \epsilon_0) \bullet ((\text{Chr 'B'} \bullet \epsilon_0) \bullet (\text{Chr 'E'} \bullet \epsilon_0))) &= \\ \text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet (\text{Chr 'E'} \bullet \epsilon_0)) & \end{aligned}$$

2.1.2 Functions as Semantic

The syntactic domain given to be the Chars language, and semantic domain now chosen to be functions over syntactic domain itself, the next step for defining a NBE algorithm is defining an evaluation function:

$$\begin{aligned} \llbracket _ \rrbracket : \text{Chars} &\rightarrow (\text{Chars} \rightarrow \text{Chars}) \\ \llbracket \epsilon_0 \rrbracket &= \text{id} \end{aligned}$$

$$\begin{aligned} \llbracket \text{Chr } c \rrbracket &= \lambda N \rightarrow (\text{Chr } c) \bullet N \\ \llbracket M \bullet N \rrbracket &= \llbracket M \rrbracket \circ \llbracket N \rrbracket \end{aligned}$$

Evaluation defined above is a simple mapping from Chars terms to functions from Chars to Chars, where empty string is mapped to identity function, singleton string to a function that concatenates the same singleton string to its input, and concatenation of strings to function composition. For instance, the two example terms representing “NBE” earlier are evaluated to the function

$$\lambda N \rightarrow \text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet (\text{Chr 'E'} \bullet N))$$

Above evaluation is also compositional.

The next step is to define a reification process:

$$\downarrow : (\text{Chars} \rightarrow \text{Chars}) \rightarrow \text{Chars}$$

$$\downarrow f = f \epsilon_0$$

Reification defined above is very simple: it applies semantic function to empty string. For example, the function

$$\lambda N \rightarrow \text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet (\text{Chr 'E'} \bullet N))$$

from earlier is reified to the following term:

$$\text{Chr 'N'} \bullet (\text{Chr 'B'} \bullet (\text{Chr 'E'} \bullet \epsilon_0))$$

Reification is also obviously compositional.

Normalisation function is defined as usual. However, the normalisation process based on function semantics is more efficient compared to the one based on list semantics. Essentially, the evaluation process using the semantic domain $\text{Char} \rightarrow \text{Char}$, evaluates terms using an efficient representation of lists, known as Hughes lists (Hughes 1986).

2.1.3 Observation

For this example, three domains are explicitly discussed: Chars syntactic domain, semantic domain based on normal lists and semantic domain based on Hughes lists. There is also a fourth domain implicit in the discussion: the syntactic domain of canonical forms, which is a subset of the syntactic domain. For Chars language, terms in canonical form are of the following grammar:

$$\begin{aligned} c &\in \text{Char (set of characters)} \\ N &\in \text{CanonicalChars} ::= \epsilon_0 \mid \text{Chr } c \bullet N \end{aligned}$$

For instance, the example canonical form derived earlier follows the above grammar.

Compared to Chars, the grammar of canonical forms is less flexible but more compact: it is easier to program in Chars, but it is also harder to analyse programs in Chars. At the cost of implementing a normalisation process, like the two NBE algorithms above, one can use benefits of the two languages: let programs to be written in the syntactic domain, since they are easier to write, then normalise the programs and let analysis be done on normalised programs, since they are easier to analyse. It is an important observation, which can be generalised to any language possessing canonical forms. Indeed, compilers use the same approach by transforming programs written in the flexible surface syntax to a more compact internal representation. For languages with computational content, such transformations often improve the performance of the programs. In this perspective, optimisation of terms can be viewed as normalisation of terms.

Next section puts this observation to work for EDSLs.

3. Embedding-By-Normalisation

Embedding is referred to a diverse set of techniques for implementing DSL terms, by first encoding them as terms in a host language, and then defining their semantics using the encoded terms.

Semantics of DSL terms may be defined entirely inside the host language by interpreting them in the host language’s runtime system, or partly outside the host language by compiling code and passing it to an external system.

The key selling points for embedding DSLs are to reuse the machinery available for a host language, from parser to type checker, and to integrate with its ecosystem, from editors to runtime system. EDSLs and embedding techniques that are proven successful in practice, go beyond traditional sole reuse of syntactic machinery such as parser and type-checker, and employ the evaluation mechanism of the host language to optimise the DSL terms (Axelsson *et al.* 2010; Svensson *et al.* 2011; Rompf 2012; Mainland and Morrisett 2010).

Briefly put, what these techniques provide is abstraction-without-guilt: the possibility to define layers of abstraction in EDSLs, using features available in the host language, without sacrificing the performance of final produced code. As mentioned in the previous section, an optimisation process, such as the ones used in above techniques, can be viewed as a normalisation process. So essentially, what the mentioned embedding techniques do is to perform **normalisation** of embedded terms **by** reusing the **evaluation** mechanism of the host language. As the names suggest, there is a correspondence between such embedding techniques and NBE begging to be examined:

$$\begin{array}{c} \text{optimisation of object by evaluation in host} \\ \longleftrightarrow \\ \text{normalisation of syntax by evaluation in semantic} \end{array}$$

This section investigates the correspondence, by drawing parallel between different components of the two sides. But, before doing so, the class of EDSLs under investigation should be specified.

3.1 Normalised EDSLs

Generally speaking, not every EDSL possess computational content, e.g., consider DSLs used for data description. On the other hand, a large and popular class of EDSLs possess some form of computational content. For the latter, as mentioned earlier, embedding techniques try to take full advantage of the evaluation process in the host language to optimise object terms before extracting code from them. The extracted code is passed, as data, to a back-end, which either interprets the data by directly calling foreign function interfaces (e.g., see Meijer *et al.* (2006a); Chakravarty *et al.* (2011)), or by passing it to an external compiler (e.g., see Axelsson *et al.* (2010); Sajeeth *et al.* (2013)). This class of EDSLs are referred to as *normalised EDSLs* in this paper, and they are distinguished from other EDSLs by the fact that (a) they possess computational content; (b) the object terms are optimised by using evaluation in the host language; and (c) they extract code from optimised object terms and the code is representable as data.

In general, embedding a DSL as a normalised EDSL constitutes of four components:

Object Language is the language defining the syntax of the DSL being embedded

Host Language is the language that the DSL is being embedded into

Encoding is the process of defining terms in the object language as a specific set of terms in the host language

Code Extraction is the process of deriving object code, as data, from the specific set of values (as opposed to general terms) in the host language that encode (optimised) object terms

Encoding of object terms as host terms is done in a way that the resulting values after evaluation of host terms denote optimised object terms.

For instance, the following is the four components in an embedding of Chars language:

- *Object language* is of the following grammar:

$$c \in \text{Char (set of characters)}$$

$$L, M, N \in \text{Chars} ::= \epsilon_0 \mid \text{Chr } c \mid M \bullet N$$

- *Host language* is a pure typed functional language

- *Encoding* is as follows:

$$\epsilon_0 \text{ is encoded as the host (nullary) function } \text{eps}_f = []$$

$$\text{Chr } c \text{ is encoded as the host function } \text{chr}_f \ c = [c]$$

$$M \bullet N \text{ is encoded as the host function } M \bullet_f N = M \text{ ++ } N$$

- *code extraction* is a function from list values to the datatype (of the $_d$ indexed constructors) representing the extracted code

$$\downarrow [] = \text{Eps}_d$$

$$\downarrow (c :: cs) = \text{Chr}_d \ c \bullet_d (\text{reify } cs)$$

Users of Chars EDSL write their programs using $_f$ indexed functions, and the extracted code, the $_d$ indexed data, is passed to back-end of the Chars EDSL. A simple example of such back-end would be a function that takes the code and prints the denoted string:

```
printChars : Chars_d → IO ⟨⟩
printChars Eps_d = printString ""
printChars (Chr_d c •_d N) = do printChar c
                             printChars N
```

3.2 Correspondence

Comparing embedding structure, explained in Section 3.1, with NBE structure, explained in Section 2, the correspondence is evident as follows:

NBE	⟷	EBN
Syntactic Domain	⟷	Object Language
Semantic Domain	⟷	Host Language (a subset of)
Evaluation	⟷	Encoding
Reification	⟷	Code Extraction

Viewing embedding through the lens of NBE, one can observe that many of the smart techniques for encoding object terms as host terms basically correspond to defining parts of an evaluation process that maps object terms to values (as opposed to general terms) in a subset of host language. Once one puts the two sides together, the correspondence between code extraction and reification in NBE is also not surprising. Even the name “reification” has been used by some embedding experts to refer to the code extraction process (see Gill (2014)).

This paper dubs an embedding process that follows the NBE structure as Embedding-By-Normalisation, or EBN for short. Embedding-by-normalisation is to be viewed as a general theoretical framework to study existing embedding techniques in practice, and also as a recipe on how to structure implementation of normalised EDSLs. EBN builds a bridge between theory and practice: theoretical solutions in NBE can be used to solve practical problems in embedding, and vice versa.

There are can be different approaches to perform embedding-by-normalisation. For instance, provided a back-end to process input code represented as data, embedding-by-normalisation follows the steps below:

1. The abstract syntax of the code expected by the back-end is identified. Such abstract syntax corresponds to the grammar of normal forms.
2. Semantic domain is identified as a subset of the host language.
3. Reification is identified as the process that maps terms in the semantic domain to terms in normal form, as usual.
4. Evaluation is identified as programs in the host language that map object terms to values in the semantic domain.

Above steps can be reordered. However, important observation here is that often defining syntax of normal forms and semantic domain should be prioritised over defining the interface that the end-users program in, i.e., the syntactic domain. Syntactic domain can be seen as the class of host programs that can be normalised to terms following the grammar of normal forms.

3.3 Encoding Strategies

Due to its correspondence to NBE, EBN is of mathematical nature: abstract and general. Furthermore, as there are variety of NBE algorithms, there are variety of corresponding EBN techniques. The generality and variety make it difficult to propose a concrete encoding strategy for EBN. The remainder of this section discusses some general encoding strategies based on the existing techniques including shallow embedding, final tagless embedding, deep embedding, and quoted embedding.

3.3.1 Shallow Embedding

Shallow embedding is when an interface formed by a set of functions in the host is used to represent the syntax of a DSL, and implementation of the functions as semantics. In shallow embedding, semantic of the DSL is required to be compositional (Carette *et al.* 2009; Gibbons and Wu 2014). In EBN, when encoding of object terms follows shallow embedding, the four components of EBN are as follows:

Syntactic Domain is an interface formed by a set of functions (or values) in the host

Semantic Domain is the result type of above interface

Evaluation is the overall evaluation of the implementation of the above interface in the host. In this setting, EBN’s evaluation process is also required to be compositional, and evaluation of a syntactic term is built up from the evaluation of its sub-terms.

Reification is a mapping from host values of the semantic domain type to data that implements a subset of syntactic domain interface, i.e., the subset that corresponds to the grammar of normal forms.

The Chars example in Section 3.1 is EBN with shallow encoding.

3.3.2 Final Tagless Embedding

Final tagless embedding (Carette *et al.* 2009), which is a specific form of shallow embedding, is when the shallow interface is parametric over the semantic type. In Haskell, the parametric interface is defined as a type-class, where instantiating the type-class defines semantics. Similar to shallow embedding, in final tagless embedding, evaluation is required to be compositional.

In EBN, when encoding of object terms follows final tagless embedding, the four components of EBN are as follows:

Syntactic Domain is a type-class (or a similar machinery such as modules) defining syntax in final tagless style

Semantic Domain is the type that the syntax type-class is instantiated with

Evaluation is the implementation of an instance of syntax type class. In this setting, EBN's evaluation process is also required to be compositional. An instance of syntax type-class is an algebra for folds over the syntactic language

Reification is a mapping from host values of the semantic domain type to data that implements a subset of syntactic domain interface, i.e., the subset that corresponds to the grammar of normal forms.

For instance, the following is the four components in EBN of Chars language with final tagless encoding:

- *Syntactic domain* is the following type-class declaration:

```
class CharsLike chars where
  eps_f : chars
  chr_f : Char → chars
  (•_f) : chars → chars → chars
```

- *Semantic domain* is the type List Char in a functional language with type-classes
- *Evaluation* is the following type-class instance:

```
instance CharsLike (List Char) where
  eps_f = []
  chr_f c = [ c ]
  m •_f n = m ++ n
```

- *Reification* is the following function

```
reify : List Char → Chars_d
reify [] = Eps_d
reify (c :: cs) = Chr_d c •_d (reify cs)
```

where code is defined as the following algebraic datatype

```
data Chars_d = Eps_d
              | Chr_d c
              | Chars_d •_d Chars_d
```

3.3.3 Deep Embedding

Deep embedding is when datatypes in host are used for representing the syntax of a DSL, and semantics is defined as functions (programs in general) over the syntax datatypes.

In EBN, when encoding of object terms follows deep embedding, the four components of EBN are as follows:

Syntactic Domain is a datatype

Semantic Domain is a type that the syntax datatype is transformed to

Evaluation is a function from syntax datatype to semantic domain

Reification is a mapping from host values of the semantic domain type to a datatype describing normal forms

For instance, the following is the four components in EBN of Chars language with deep encoding:

- *Syntactic domain* is the following datatype:

```
data Chars_d = Eps_d
              | Chr_d c
              | Chars_d •_d Chars_d
```

- *Semantic domain* is the type List Char in a functional language with algebraic datatypes
- *Evaluation* is the following function:

```
eval : Chars_d → List Char
eval Eps_d = []
eval (Chr_d c) = [ c ]
eval (m •_d n) = m ++ n
```

- *Reification* is the following function

```
reify : List Char → Chars_d
reify [] = Eps_d
reify (c :: cs) = Chr_d c •_d (reify cs)
```

3.3.4 Quoted Embedding

Quoted embedding (Najd *et al.* 2016), which is a specific form of deep embedding, is when some form of quotations is used to represent syntax, and semantics is defined as functions over the unquoted representation.

In EBN, when encoding of object terms follows quoted embedding, the four components of EBN are as follows:

Syntactic Domain is the type of quoted terms in the host

Semantic Domain is a type that the unquoted representation of syntactic terms is transformed to

Evaluation is a function from unquoted representation of syntactic terms to semantic domain

Reification is a mapping from host values of the semantic domain type to a datatype describing normal forms

For instance, the following is the four components in EBN of Chars language with quoted encoding:

- *Syntactic domain* is $Chars_d$, the resulting type of a quasi-quotation denoted as $[c|\dots]$ for the grammar of Chars language.
- *Semantic domain* is the type List Char in a functional language with quasi-quotation
- *Evaluation* is the following function:

```
eval : Chars_d → List Char
eval [c| ε_0 ] = []
eval [c| Chr $c ] = [ c ]
eval [c| $m • $n ] = m ++ n
```

where \$ denotes anti-quotation (Mainland 2007).

- *Reification* is the following function

```
reify : List Char → Chars_d
reify [] = [c| ε_0 ]
reify (c :: cs) = [c| Chr $c • $ (reify cs) ]
```

4. Embedding-By-Normalisation, Generically

Back in 1966, Landin in his landmark paper "The Next 700 Programming Languages" (Landin 1966) argues that seemingly different programming languages can be seen as instances of one unified language and that the differences can be factored as normal libraries for the unified language. Landin nominates lambda calculus as the unified language, and shows how to encode seemingly different language constructs as normal programs in this language. Since then, Landin's idea has been proven correct over and over again, evidenced by successful functional programming languages built based on the very idea (e.g., see the design of Glasgow Haskell Compiler).

Although Landin's idea was originally expressed in terms of general-purpose languages, it also applies to domain-specific ones. Following in his footsteps, this section considers DSLs which can be expressed using the lambda calculus enriched with primitive

values and operations (Plotkin 1975) to express domain-specific constructs. Not all DSLs can be modelled in this way, and the principles of EBN applies even outside this model. But this model covers a large and useful class of DSLs and allows for a parametric presentation of DSLs, where syntax of a DSL can be identified solely by the signature of the primitive values and operations.

4.1 Simple Types and Products

This subsection presents an instantiations of the EBN technique for DSLs which can be captured by the simply-typed lambda calculus with product types, parametric over the set of base types, literals, and the signature of primitive operations. The host language is assumed to be a pure typed functional language, and the subset the EBN technique is targeting (i.e., the semantic domain) is identified by a constraint on the type of host terms.

4.1.1 Syntactic Domain

The grammar of types in the object language is standard:

$$\begin{aligned} \chi &\in X \text{ (set of base types)} \\ A, B &\in \text{Type} ::= \underline{\chi} \mid \underline{\langle \rangle} \mid A \Rightarrow B \mid A \times B \end{aligned}$$

It is parametric over a set of base types. Besides base types, it includes unit, function type, and product type. The types of the object language are underlined to distinguish it from the ones of the host language.

The grammar of the terms in the object language is also standard (Filinski 1999):

$$\begin{aligned} x &\in \Gamma \text{ (set of variables)} \\ \xi &\in \Xi \text{ (set of literals)} \\ c &\in \Sigma \text{ (set of signature of primitives)} \\ L, M, N &\in \text{Syn} ::= \underline{\xi} \mid \underline{c \overline{M}} \mid \underline{\langle \rangle} \mid x \mid \underline{\lambda x \Rightarrow N} \mid \underline{L @ M} \\ &\quad \mid \underline{(M, N)} \mid \underline{\text{fst } L} \mid \underline{\text{snd } L} \end{aligned}$$

The language, referred to as Syn, is parametric over a set of literals, and signature of primitive operations. Besides literals, and primitive operations (which are assumed to be fully applied), it involves unit term, variables, lambda abstraction, application, pairs, and projections. The terms of the object language are underlined to distinguish it from the ones of the host language. The typing rules are the expected ones:

$$\begin{aligned} &\frac{\xi \in \Xi_T \chi}{\Gamma_T \vdash \underline{\xi} : \underline{\chi}} && \frac{(x : A) \in \Gamma_T}{\Gamma_T \vdash x : A} \\ &\frac{\Gamma_T \vdash \underline{M_i} : A_i \quad (c : [\dots, A_i, \dots] \mapsto B) \in \Sigma_T}{\Gamma_T \vdash c \overline{M} : B} \\ &\frac{\Gamma_T, x : A \vdash N : B}{\Gamma_T \vdash \underline{\lambda x \Rightarrow N} : A \Rightarrow B} && \frac{\Gamma_T \vdash L : A \Rightarrow B \quad \Gamma_T \vdash M : A}{\Gamma_T \vdash L @ M : B} \\ &\frac{}{\Gamma_T \vdash \underline{\langle \rangle} : \underline{\langle \rangle}} && \frac{\Gamma_T \vdash M : A \quad \Gamma_T \vdash N : B}{\Gamma_T \vdash \underline{(M, N)} : A \times B} \\ &\frac{\Gamma_T \vdash L : A \times B}{\Gamma_T \vdash \underline{\text{fst } L} : A} && \frac{\Gamma_T \vdash L : A \times B}{\Gamma_T \vdash \underline{\text{snd } L} : B} \end{aligned}$$

In this section, including the other subsections, the EBN technique is presented in a way that it is independent of encoding strategy: the underlined terms can be trivially encoded using the standard methods, such as the ones explained in Section 3.3. One possible difficulty might be the treatment of free variables, which can be trivially addressed by using well-known techniques such as Higher-Order Abstract Syntax (HOAS) representation

(Pfenning and Elliott 1988). Representation of object terms is assumed be quotient with respect to alpha-conversion, when new bindings are introduced variables are assumed to be fresh, and substitutions to be capture avoiding. Terms in syntactic domain and the normal syntactic terms are represented in the same way. Though in practice, depending on encoding, the two may be implemented in different ways. In this paper, host programs of the type Syn A refer to both the type of a host term encoding a term of the type A in syntactic domain, and the type of the extracted code for a normal syntactic term of the type A. The type Syn Γ_T A denotes open Syn A terms with Γ_T being a typing environment containing the free variables. Literals in the object language, denoted as $\underline{\xi}$, are a subset of the ones in the host identified by the set Ξ . To retrieve a corresponding literal value in the host language, the underline notation is removed. Ξ_T is a mapping from base types in the object language to types in the host language. $\xi \in \Xi_T \chi$ denotes a predicate asserting that the literal ξ is an element the type $\Xi_T \chi$ in the host language. Σ is a mapping from the name of a primitive operation to its arity. Primitive operations are fully applied, a series of arguments, or a series of typing judgements relating to a primitive is denoted by overlining. Σ_T is the typing environment for primitive operations, with elements of the form $c : [A_0, \dots, A_n] \mapsto B$ which reads as that the primitive c takes $n + 1$ (arity of the primitive) elements of the type A_i , for i ranging from 0 to n , and returns a value of type B .

Now that the syntactic domain has been defined, i.e., the Syn language, it is time to define the semantic domain.

4.1.2 Semantic Domain

As the host language is a pure typed functional language, and the object language being a tiny pure typed functional language itself, a considerable part of the object language closely mirrors the one of the host language. Moreover, the representation of the syntactic domain itself, is a program in the host language, i.e., a term of the type Syn A. This observation is realised by defining semantic domain as follows:

$$\begin{aligned} \text{Sem } A &= \forall (\alpha : \text{Type}) \rightarrow \alpha \sim A \Rightarrow \alpha \\ &\frac{}{\text{Syn } A \sim A} \text{Syn}_r && \frac{}{\langle \rangle \sim \langle \rangle} \langle \rangle_r \\ &\frac{\alpha \sim A \quad \beta \sim B}{(\alpha \rightarrow \beta) \sim (A \Rightarrow B)} \rightarrow_r && \frac{\alpha \sim A \quad \beta \sim B}{(\alpha \times \beta) \sim (A \times B)} \times_r \end{aligned}$$

That is, a term of type A in the semantic domain is any host term whose type respects the \sim relation. The relation \sim states that (a) semantic terms of unit, function, and product type correspond to host terms of similar type, (b) a syntactic term encoded in the host directly correspond to a semantic term of the same type. Condition (b) is a distinguishing feature in the definition of evaluation and semantic domain of NBE. As mentioned in Section 2, evaluation in NBE is allowed to leave parts of syntactic terms uninterpreted. An uninterpreted part is referred to as a residualised part, and the act of leaving a part uninterpreted as residualising.

4.1.3 Evaluation

Except for terms of base type, evaluation process is standard: syntactic terms are mapped to corresponding host terms. Terms of base types, however, are residualised. The definition of evaluation function is as follows:

$$\begin{aligned} \llbracket _ \rrbracket &: \text{Type} \rightarrow \text{Type} \\ \llbracket \underline{\chi} \rrbracket &= \text{Syn } \chi \\ \llbracket \underline{\langle \rangle} \rrbracket &= \langle \rangle \end{aligned}$$

$$\begin{aligned} \llbracket A \Rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket \\ \llbracket A \times B \rrbracket &= \llbracket A \rrbracket \times \llbracket B \rrbracket \end{aligned}$$

$$\llbracket _ \rrbracket : \text{Syn } \Gamma_T A \rightarrow \llbracket \Sigma_T \rrbracket \rightarrow \llbracket \Gamma_T \rrbracket \rightarrow \llbracket A \rrbracket$$

$$\begin{aligned} \llbracket \xi \rrbracket \Sigma_V \Gamma_V &= \xi \\ \llbracket c \overline{M} \rrbracket \Sigma_V \Gamma_V &= \Sigma_V c \llbracket \overline{M} \rrbracket \\ \llbracket \langle \rangle \rrbracket \Sigma_V \Gamma_V &= \langle \rangle \\ \llbracket x \rrbracket \Sigma_V \Gamma_V &= \Gamma_V x \\ \llbracket \lambda x \Rightarrow N \rrbracket \Sigma_V \Gamma_V &= \lambda y \rightarrow \llbracket N \rrbracket \Sigma_V (\Gamma_V, x \mapsto y) \\ \llbracket L @ M \rrbracket \Sigma_V \Gamma_V &= (\llbracket L \rrbracket \Sigma_V \Gamma_V) (\llbracket M \rrbracket \Sigma_V \Gamma_V) \\ \llbracket (M, N) \rrbracket \Sigma_V \Gamma_V &= (\llbracket M \rrbracket \Sigma_V \Gamma_V, \llbracket N \rrbracket \Sigma_V \Gamma_V) \\ \llbracket \text{fst } L \rrbracket \Sigma_V \Gamma_V &= \text{fst } (\llbracket L \rrbracket \Sigma_V \Gamma_V) \\ \llbracket \text{snd } L \rrbracket \Sigma_V \Gamma_V &= \text{snd } (\llbracket L \rrbracket \Sigma_V \Gamma_V) \end{aligned}$$

Apart from the input expression, the evaluation function takes two extra arguments: variable Σ_V of type $\llbracket \Sigma_T \rrbracket$, that is the environment of semantic values corresponding to each primitive operator; and variable Γ_V of type $\llbracket \Gamma_T \rrbracket$, that is the environment of semantic values corresponding to each free variable. Following the convention, the semantic bracket notation is overloaded, and denotes the mapping of different kinds of elements from syntax to semantic.

4.1.4 Reification

The final step is to define the reification function. Reification can be defined as a function indexed by the relation between syntax and semantics:

$$\begin{aligned} \downarrow : \alpha \sim A \rightarrow \alpha \rightarrow \text{Syn } A \\ \downarrow \text{Syn}_r \quad V &= V \\ \downarrow \langle \rangle_r \quad V &= \langle \rangle \\ \downarrow (a \rightarrow_r b) V &= \lambda x \Rightarrow \downarrow b (V (\uparrow a \times)) \\ \downarrow (a \times_r b) V &= (\downarrow a (\text{fst } V)) \downarrow \downarrow b (\text{snd } V) \\ \uparrow : \alpha \sim A \rightarrow \text{Syn } A \rightarrow \alpha \\ \uparrow \text{Syn}_r \quad M &= M \\ \uparrow \langle \rangle_r \quad M &= \langle \rangle \\ \uparrow (a \rightarrow_r b) M &= \lambda x \rightarrow \uparrow b (M @ (\downarrow a \times)) \\ \uparrow (a \times_r b) M &= (\uparrow a (\text{fst } M), \uparrow b (\text{snd } M)) \end{aligned}$$

Above definition is similar to some classic NBE algorithms such as Berger and Schwichtenberg (1991) and Danvy (1996b). Essentially, what above does is a form of η -expansion in two levels: object language and host language (Danvy 1996b). The reification function \downarrow is mutually defined with the function \uparrow referred to as the reflection function.

Embedding a term by the EBN algorithm defined in this subsection results in a code for the corresponding term in η -long β -normal form. However, the normal form is not extensional, in that two normal terms may be equivalent but syntactically distinct (see Section 4.4).

4.2 Sums

This section extends the EBN technique of the previous subsection, to support DSL programs involving sum types, such as conditional expressions.

4.2.1 Syntactic Domain

The grammar of the syntactic domain in Section 4.1.1 is extended as follows:

$$\begin{aligned} A, B ::= \dots \mid A \pm B \\ L, M, N ::= \dots \mid \text{inr } M \mid \text{inr } N \mid \text{case } L \text{ M } N \end{aligned}$$

$$\begin{aligned} &\dots \\ \frac{\Gamma_T \vdash M : A}{\Gamma_T \vdash \text{inl } M : A \pm B} & \quad \frac{\Gamma_T \vdash N : B}{\Gamma_T \vdash \text{inr } N : A \pm B} \\ \frac{\Gamma_T \vdash L : A \pm B \quad \Gamma_T \vdash M : A \Rightarrow C \quad \Gamma_T \vdash N : B \Rightarrow C}{\Gamma_T \vdash \text{case } L \text{ M } N : C} \end{aligned}$$

The extensions are standard: sum types, left injection, right injection, and case expression. To simplify the presentation, branches of the case expression $\text{case } L \text{ M } N$ (i.e., M and N) are standard terms of function type, as opposed to a specific built-in language constructs with bindings. It follows Alonzo Church's original idea that all variable bindings in syntax can be done via bindings in lambda abstractions.

4.2.2 Semantic Domain

To support sum types, it is not enough to simply add a clause to the relation \sim of Section 4.1.2 relating sum types in the host to the ones in the object. Treating sum types has been a challenging problem in NBE and embedding. Essentially, to reify a semantic term of the type $(\text{Syn } A + \text{Syn } B) \rightarrow \text{Syn } C$, following the same symmetric style of reify-reflect process in Section 4.1.4, one needs (due to contravariance of function type) to convert a syntactic term of the type $\text{Syn } (A \pm B)$ to a semantic term of the type $\text{Syn } A + \text{Syn } B$. The conversion of the type $\text{Syn } (A \pm B) \rightarrow \text{Syn } A + \text{Syn } B$ is problematic, since there is no way to destruct a term of the type Syn and remain in the semantic domain; the output type of the function is a term in the semantic domain, while destructing a sum type in syntactic domain demands a continuation in the syntactic domain. For a more detailed account of the reification problem for sum types, refer to Najd *et al.* (2016); Gill (2014); Svenningsson and Axelsson (2015). In the context of type-directed partial evaluation, a NBE based technique, Danvy (1996b) proposed an elegant solution to the problem of reification of sums, using composable continuations (delimited continuations) based on shift and reset (Danvy and Filinski 1990). This paper employs Danvy's solution.

Delimited continuations are effect-full constructs. To model them in the pure and typed setting of the host language, this paper uses the standard monadic semantic (e.g., see (Atkey 2009; Dybvig *et al.* 2007; Wadler 1994)). The \sim relation from Section 4.1.2 is updated as follows:

$$\begin{aligned} &\dots \\ \frac{\alpha \sim A \quad \beta \sim B}{(\alpha \rightsquigarrow \beta) \sim (A \Rightarrow B)} \rightarrow_r \quad \frac{\alpha \sim A \quad \beta \sim B}{(\alpha + \beta) \sim (A \pm B)} +_r \end{aligned}$$

where \rightsquigarrow denotes type of monadic functions, i.e., effect-full functions modelled in the mentioned standard monadic semantic.

One subtle, yet important factor in play here is the perspective that EBN offers: Danvy's elegant use of shift and reset is not a mere technical solution (even if it may seem like so when used in an untyped impure language); through the lens of NBE/EBN, it can be seen as a change of semantic domain to a monadic one, where the use of shift and reset are the resulting consequences.

4.2.3 Evaluation

Since we will use a monad for delimited continuations the semantic domain will be updated so that functions in the object language are now mapped to monadic functions in the host language. As anticipated, we also add sums, which maps to sums in the host language.

$$\begin{aligned} \dots \\ \llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightsquigarrow \llbracket B \rrbracket \\ \llbracket A \pm B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \end{aligned}$$

The evaluator now needs to be updated to reflect the fact that the semantic domain uses monadic functions. All of the cases from the evaluator Section 4.1.3 have been updated to lift the result into a monad.

$$\begin{aligned} \llbracket _ \rrbracket : \text{Syn } \Gamma_T A \rightarrow \llbracket \Sigma_T \rrbracket \rightarrow \llbracket \Gamma_T \rrbracket \rightarrow \llbracket A \rrbracket \\ \llbracket \xi \rrbracket \Sigma_V \Gamma_V &= \langle \xi \rangle \\ \llbracket c \ \overline{M} \rrbracket \Sigma_V \Gamma_V &= \Sigma_V c \ \llbracket \overline{M} \rrbracket \\ \llbracket \langle \rangle \rrbracket \Sigma_V \Gamma_V &= \langle \rangle \\ \llbracket x \rrbracket \Sigma_V \Gamma_V &= \Gamma_V x \\ \llbracket \lambda x \rightarrow N \rrbracket \Sigma_V \Gamma_V &= \langle \lambda y \rightarrow \llbracket N \rrbracket \Sigma_V (\Gamma_V, x \mapsto \langle y \rangle) \rangle \\ \llbracket L @ M \rrbracket \Sigma_V \Gamma_V &= \text{join } \langle \llbracket L \rrbracket \Sigma_V \Gamma_V, \langle \llbracket M \rrbracket \Sigma_V \Gamma_V \rangle \rangle \\ \llbracket (M, N) \rrbracket \Sigma_V \Gamma_V &= \langle \langle \llbracket M \rrbracket \Sigma_V \Gamma_V, \llbracket N \rrbracket \Sigma_V \Gamma_V \rangle \rangle \\ \llbracket \text{fst } L \rrbracket \Sigma_V \Gamma_V &= \langle \text{fst } (\llbracket L \rrbracket \Sigma_V \Gamma_V) \rangle \\ \llbracket \text{snd } L \rrbracket \Sigma_V \Gamma_V &= \langle \text{snd } (\llbracket L \rrbracket \Sigma_V \Gamma_V) \rangle \\ \llbracket \text{inl } M \rrbracket \Sigma_V \Gamma_V &= \langle \text{inl } (\llbracket M \rrbracket \Sigma_V \Gamma_V) \rangle \\ \llbracket \text{inr } N \rrbracket \Sigma_V \Gamma_V &= \langle \text{inr } (\llbracket N \rrbracket \Sigma_V \Gamma_V) \rangle \\ \llbracket \text{case } L \ M \ N \rrbracket \Sigma_V \Gamma_V &= \text{join } \langle \text{case } (\llbracket L \rrbracket \Sigma_V \Gamma_V) \\ &\quad (\llbracket M \rrbracket \Sigma_V \Gamma_V) \\ &\quad (\llbracket N \rrbracket \Sigma_V \Gamma_V) \rangle \end{aligned}$$

For clarity of presentation, applicative bracket notation (McBride and Paterson 2008) is used in above (denoted as $\langle \dots \rangle$). An applicative bracket notation $\langle L \ M_0 \dots M_n \rangle$ is a mere syntactic sugar equivalent to the following term using monadic do notation:

```
do x0 ← M0
...
xn ← Mn
return (L x0 ... xn)
```

The function join is a the well-known monad join function, commonly used for flattening nested monadic structures.

4.2.4 Reification

To adopt Danvy's solution, the Reification process of 4.1.4 is updated as follows:

$$\begin{aligned} \dots \\ \downarrow (a \rightarrow_r b) \ V &= \lambda x \rightarrow \text{reset } \langle \downarrow b \ (\text{join } \langle V \ (\uparrow a \ x) \rangle) \rangle \\ \downarrow (a +_r b) \ V &= \text{case } V \ (\lambda x \rightarrow \text{inl } \langle \downarrow a \ x \rangle) \\ &\quad (\lambda y \rightarrow \text{inr } \langle \downarrow b \ y \rangle) \\ \uparrow : \alpha \rightsquigarrow A \rightarrow \text{Syn } A \rightsquigarrow \alpha \\ \uparrow \text{Syn}_r \quad M &= \langle M \rangle \\ \uparrow \langle \rangle_r \quad M &= \langle \langle \rangle \rangle \\ \uparrow (a \rightarrow_r b) \ M &= \langle \lambda x \rightarrow \uparrow b \ (M @ \langle \downarrow a \ x \rangle) \rangle \\ \uparrow (a \times_r b) \ M &= \langle \langle \uparrow a \ (\text{fst } M), \uparrow b \ (\text{snd } M) \rangle \rangle \\ \uparrow (a +_r b) \ M &= \text{shift } (\lambda k \rightarrow \\ &\quad \text{case } M \ (\lambda x \rightarrow \text{reset } \langle (k \circ \text{inl}) \ (\uparrow a \ x) \rangle) \\ &\quad (\lambda y \rightarrow \text{reset } \langle (k \circ \text{inr}) \ (\uparrow b \ y) \rangle)) \end{aligned}$$

Except for the necessary monadic liftings, the nontrivial change is the clause related to sum types in the reflection function. Instead of destructing M directly, which results in another term of Syn type, first it asks for a continuation k that given a semantic value of sum type, produces a syntactic term, and then uses this continuation for constructing the syntactic continuations needed for destructing M . Reader interested in more details, may try to follow above algorithm step-by-step to reify the semantic term $\lambda x \rightarrow \langle \rangle$ of the type $(\text{Syn } \langle \rangle + \text{Syn } \langle \rangle) \rightarrow \text{Syn } \langle \rangle$, or the term $\lambda x \rightarrow \text{fst } x$ of the same type, and consult Danvy (1996b), if needed.

4.3 Smart Primitives

So far, syntactic terms of base types have been residualised: they are treated as uninterpreted entities. This subsection proposes an alternative semantic domain, so that some of the primitives can be mapped to their corresponding host programs and terms with syntactic terms with base types get partially normalised.

4.3.1 Syntactic Domain

Syntactic domain in this subsection is the same as the one in Section 4.2.1.

4.3.2 Semantic Domain

The type relation \sim in 4.2.2 does not consider convertibility of semantic values in the host language: given object type A , if value V in the host is convertible, by a function in the host, to another value W which respects $\sim A$, V also respects $\sim A$. The following is a generalisation of the type relation \sim in 4.2.2, based on this observation:

$$\begin{aligned} \frac{}{\text{Syn } A \sim^p A} \text{Syn}_r \quad \frac{}{\langle \rangle \sim^p \langle \rangle} \langle \rangle_r \\ \frac{(\alpha \sim^p A) \ (\beta \sim^p B)}{(\alpha \rightsquigarrow \beta) \sim^p (A \rightarrow B)} \rightarrow_r \\ \frac{(\alpha \sim^p A) \ (\beta \sim^p B)}{(\alpha \times \beta) \sim^p (A \times B)} \times_r \quad \frac{(\alpha \sim^p A) \ (\beta \sim^p B)}{(\alpha + \beta) \sim^p (A \pm B)} +_r \\ \frac{(\alpha \sim^p A) \ (\exists f : \alpha \rightarrow \beta)}{\beta \sim^p A} \uparrow_r \quad \frac{(\beta \sim^p A) \ (\exists f : \alpha \rightarrow \beta)}{\alpha \sim^p A} \uparrow_r^+ \end{aligned}$$

Notice that the polarity, or variance, of a type should be tracked as the type of conversion functions changes direction due to contravariance of arguments in function types. For instance, consider function f converting $V : \text{Syn } A \rightarrow \text{Syn } B$ to $f \ V : (\text{Syn } A, \text{Syn } A) \rightarrow \text{Syn } B$. One should provide a function g of type $(\text{Syn } A, \text{Syn } A) \rightarrow \text{Syn } A$, rather than $\text{Syn } A \rightarrow (\text{Syn } A, \text{Syn } A)$, so that $f \ V = \lambda x \rightarrow V \ (g \ x)$.

4.3.3 Evaluation

Evaluation is similar to the one in Section 4.2.3, except that here base types can be either residual syntactic terms, as before, or the corresponding values in the semantic domain:

$$\llbracket \chi \rrbracket = \text{Syn } \chi + \Xi_T \chi$$

$$\llbracket \xi \rrbracket \Sigma_V \Gamma_V = \langle (\text{inr } \xi) \rangle$$

This rather simple change has a practically significant impact: primitive operations defined in Σ_V , can now pattern match on their input of base type, and provide optimised versions based on the available values. This is demonstrated in the example presented in Section 4.5. For clarity, the following datatype can be used instead of plain sums:

```
data PossibleValue  $\chi$  = Exp (Syn  $\chi$ )
                    | Val ( $\Xi_T \chi$ )
```

4.3.4 Reification

Reification in Section 4.2.4, is updated to take into account the convertibility rules:

$$\begin{aligned} \downarrow : \alpha \rightsquigarrow^+ A \rightarrow \alpha \rightarrow \text{Syn } A \\ \downarrow \dots \end{aligned}$$

$$\begin{aligned}
\downarrow (\uparrow^+ a f) V &= \downarrow a (f V) \\
\uparrow : \alpha \sim A &\rightarrow \text{Syn } A \rightsquigarrow \alpha \\
\uparrow \dots & \\
\uparrow (\uparrow^+ a f) M &= \langle f (\uparrow a M) \rangle
\end{aligned}$$

4.4 Richer Languages

There is a wealth of solutions available in NBE and related areas that can be adopted to support embedding by normalisation of languages with features not covered in this paper.

In the context of partial evaluation, Danvy, his collaborators, and others worked on a series of extensions to the algorithm presented above. They considered support for object language features such as recursion, side-effects, syntactic sharing, laziness and memoization, and datatypes (e.g., see Danvy (1996a); Balat and Danvy (2002); Danvy (1998); Sheard (1997)).

In the context of type theory, Altenkirch, Dybjer, and others extended NBE to richer type systems. They considered systems such as variants of Martin-Löf type theory (e.g., see Abel *et al.* (2007)), polymorphic lambda calculus (Altenkirch and Uustalu 2004), and simply typed lambda calculus with strong sums (Altenkirch *et al.* 2001; Balat *et al.* 2004).

4.5 An Example

As an example, consider the "hello world" of program generation, the power function:

$$\begin{aligned}
\text{power} &: \mathbb{Z} \rightarrow \text{Syn } (\mathbb{Q} \rightarrow \mathbb{Q}) \\
\text{power } n &= \lambda x \rightarrow \\
&\quad \text{if } n < 0 \text{ then} \\
&\quad \quad \text{if } x \equiv 0 \\
&\quad \quad \quad \text{then } 0 \\
&\quad \quad \quad \text{else } (-1 / (\text{power } (-n) @ x)) \\
&\quad \text{else if } n == 0 \text{ then} \\
&\quad \quad 1 \\
&\quad \text{else if even } n \text{ then} \\
&\quad \quad (\text{let } y = \text{power } (n / 2) @ x \\
&\quad \quad \quad \text{in } y * y) \\
&\quad \text{else} \\
&\quad x * (\text{power } (n - 1) @ x)
\end{aligned}$$

It takes two arguments and raises the second to the power of the first argument. Following the convention (e.g., see Najd *et al.* (2016)), it is written in the "staged" style: power is a meta-function in the host language that provided integer host value n produces object terms of the type $\mathbb{Q} \rightarrow \mathbb{Q}$. For pedagogical purposes, we avoid techniques that further optimise this function but obscure its presentation.

Following the parametric model proposed in this section, for defining the syntax one only needs to provide the following:

Base Types, which includes type rational numbers

$$X = \{\mathbb{Q}\}$$

Literals, which includes literals of rational numbers

$$\Xi_T = \{\mathbb{Q} \mapsto \mathbb{Q}\}$$

Primitives, which includes equality, multiplication, and division operations on rational numbers

$$\begin{aligned}
\Sigma &= \{ \equiv \mapsto 2, \\
&\quad * \mapsto 2, \\
&\quad / \mapsto 2 \} \\
\Sigma_T &= \{ \equiv : \{\mathbb{Q}, \mathbb{Q}\} \mapsto \text{Bool}, \\
&\quad * : \{\mathbb{Q}, \mathbb{Q}\} \mapsto \mathbb{Q}, \\
&\quad / : \{\mathbb{Q}, \mathbb{Q}\} \mapsto \mathbb{Q} \}
\end{aligned}$$

$$\begin{aligned}
/ &: \{\mathbb{Q}, \mathbb{Q}\} \mapsto \mathbb{Q} \\
\Sigma_V &: \{\Sigma_T\} \\
\Sigma_V &= \{ \{ \\
&\quad (\text{Val } V) ==_v (\text{Val } W) = \langle V == W \rangle \\
&\quad (\text{Val } V) ==_v (\text{Exp } N) = \uparrow \text{Bool}_r (\underline{V} \equiv N) \\
&\quad (\text{Exp } M) ==_v (\text{Val } W) = \uparrow \text{Bool}_r (M \equiv \underline{W}) \\
&\quad (\text{Exp } M) ==_v (\text{Exp } N) = \uparrow \text{Bool}_r (M \equiv N), \\
&\quad (\text{Val } V) *_v (\text{Val } W) = \langle \text{Val } (V * W) \rangle \\
&\quad (\text{Val } 1) *_v (\text{Exp } N) = \langle \text{Exp } N \rangle \\
&\quad (\text{Val } V) *_v (\text{Exp } N) = \langle \text{Exp } (\underline{V} * N) \rangle \\
&\quad (\text{Exp } M) *_v (\text{Val } 1) = \langle \text{Exp } M \rangle \\
&\quad (\text{Exp } M) *_v (\text{Val } W) = \langle \text{Exp } (M * \underline{W}) \rangle \\
&\quad (\text{Exp } M) *_v (\text{Exp } N) = \langle \text{Exp } (M * N) \rangle, \\
&\quad (\text{Val } V) /_v (\text{Val } W) = \langle \text{Val } (V / W) \rangle \\
&\quad (\text{Val } V) /_v (\text{Exp } N) = \langle \text{Exp } (\underline{V} / N) \rangle \\
&\quad (\text{Exp } M) /_v (\text{Val } 1) = \langle \text{Exp } M \rangle \\
&\quad (\text{Exp } M) /_v (\text{Val } W) = \langle \text{Exp } (M / \underline{W}) \rangle \\
&\quad (\text{Exp } M) /_v (\text{Exp } N) = \langle \text{Exp } (M / N) \rangle \} \}
\end{aligned}$$

Above relies on the definition of Boolean values defined as a sum of unit types:

$$\begin{aligned}
\text{Bool} &= \langle \rangle \pm \langle \rangle \\
\text{false} &= \text{inl } \langle \rangle \\
\text{true} &= \text{inr } \langle \rangle \\
\text{if } L \text{ then } M \text{ else } N &= \text{case } L (\lambda x \rightarrow N) (\lambda y \rightarrow M) \\
\text{Bool}_r &= \langle \rangle_r +_r \langle \rangle_r
\end{aligned}$$

Running norm (power -6) results in the following code:

$$\begin{aligned}
(\lambda x_0 \rightarrow \text{if } (x_0 == 0) \\
&\quad \text{then } 0 \\
&\quad \text{else } (-1 / ((x_0 * (x_0 * x_0)) * \\
&\quad \quad (x_0 * (x_0 * x_0))))))
\end{aligned}$$

Notice that primitives in Σ_V are smart. They are defined by pattern matching on the inputs, and produce optimised terms based on the available inputs. For instance, multiplication of a syntactic term M by one, simplifies to M . Without such smart primitives, running norm (power -6) results in the following code:

$$\begin{aligned}
(\lambda x_0 \rightarrow \text{if } (x_0 == 0) \\
&\quad \text{then } 0 \\
&\quad \text{else } (-1 / ((x_0 * ((x_0 * 1) * (x_0 * 1))) * \\
&\quad \quad (x_0 * ((x_0 * 1) * (x_0 * 1)))))
\end{aligned}$$

To demonstrate normalisation of sum types, a simple form of abstraction is considered: handling corner-cases. The definition of Power is split into two parts. One alters definition of Power to return nothing of Maybe type instead of 0 when division by zero happens, and another replaces nothing by 0:

$$\begin{aligned}
\text{power}' &: \mathbb{Z} \rightarrow \text{Syn } (\mathbb{Q} \rightarrow \text{Maybe } \mathbb{Q}) \\
\text{power}' n &= \lambda x \rightarrow \\
&\quad \text{if } n < 0 \text{ then} \\
&\quad \quad \text{if } x \equiv 0 \\
&\quad \quad \quad \text{then nothing} \\
&\quad \quad \quad \text{else } ((\lambda y \rightarrow -1 / y) \\
&\quad \quad \quad \quad \langle \$ \rangle (\text{power}' (-n) @ x)) \\
&\quad \text{else if } n == 0 \text{ then} \\
&\quad \quad \text{just } 1 \\
&\quad \text{else if even } n \text{ then} \\
&\quad \quad ((\lambda y \rightarrow y * y) \\
&\quad \quad \quad \langle \$ \rangle (\text{power}' (n / 2) @ x))
\end{aligned}$$

```

else
  ((λ y ⇒ x * y)
   ⟨$⟩ (power' (n - 1) @ x))
power'' : ℤ → Syn (Q ⇒ Q)
power'' n = λ x ⇒ maybe (λ z ⇒ z) 0 (power' n @ x)

```

Above relies on the definition of Maybe values of rational numbers defined as a sum type:

```

MaybeQ      = Q ± ⟨⟩
just x       = inl x
nothing      = inr ⟨⟩
maybe M N L = case L (λ x ⇒ M @ x) (λ y ⇒ N)
L ⟨$⟩ M      = maybe (λ x ⇒ just (L @ x)) nothing M

```

Running norm (power'' -6) results in exactly the same value as norm (power -6). Normalisation removes the unnecessary code in power'', and makes it behave as if no additional layer of abstraction has been introduced in the first place; as demonstrated, normalisation in EBN achieves abstraction-without-guilt, even for sum types.

Besides the implementation in Agda, examples and the corresponding EBN technique in this subsection is implemented in Haskell and is available at <https://github.com/shayan-najd/Embedding-By-Normalisation/blob/master/Power.hs>.

5. Discussion & Related Work

As with any other paper describing correspondence between two areas, this paper introduces the main body of the related works, while explaining the related concepts. Focus of this section is to mention key related areas, besides NBE, and where EBN stands in comparison.

5.1 Normalised EDSLs

The work by Svenningsson and Axelsson (2015) is perhaps the most closely related to what is presented in this paper. They provide a way to embed languages which combines deep and shallow embeddings which allows DSLs to be normalised by using evaluation in the host language. Phrased in the framework of Embedding by Normalisation, their methodology matches the form of embedding presented in section 4.1. They limit their system to a first-order fragment, to produce efficient and computationally predictable C code. They can use host language functions and products for their DSL implementations. Though they cannot deal with arbitrary sum types, although they provide tricks for dealing with a restricted form of sum type, such as the Maybe type in the Haskell standard library. Their implementation in Haskell uses a type-class which contains two methods for converting from shallow embedding to deep embedding of terms and vice versa. The type-class and its instances correspond to the reification function in EBN, where conversion from shallow to deep can be seen as reification function and conversion from deep to shallow as reflection function. Examples of DSLs which use this style of embedding are Feldspar (Axelsson *et al.* 2010), Obsidian (Svensson *et al.* 2011) and Nikola (Mainland and Morrisett 2010).

Other important related works, are series of successful EDSLs implemented in Scala using LMS (Rompf and Odersky 2010; Rompf 2012). They use evaluation mechanism of the host language for optimising DSLs. Their systems are based on a form staged computation (e.g., see Taha and Sheard (2000)), and they do so by a smart type-directed approach utilising virtualisation (e.g., see Rompf *et al.* (2013)). Rompf has characterised the essence of LMS, as an approach based on the two-level lambda calculi (e.g., see (Nielson and Nielson 1992)). As explored by Danvy (1996b), NBE and two-level calculi are related. One interesting future work is to exploit the relation to compare EBN with the technique underlying LMS.

There are also large bodies of works on embedding specific DSLs in Haskell that use the evaluation mechanism of Haskell to optimise embedded terms. Gill (2014) provides a general overview of embedding techniques in Haskell, and a crisp explanation of the reification problems addressed in this paper. One possible explanation of why reification for sum types or primitives appeared difficult (if not impossible), is that DSL designers presumed the semantic domain to be a simple sub-set of the host language without continuations or lifted base types, i.e., the one in Section 4.1.2. As EBN reveals, to be able to reify terms involving sums or primitives, one needs to settle for an alternative semantic domain.

5.2 Partial Evaluation

As mentioned in Section 4.4, Danvy's Type-Directed Partial Evaluation (Danvy 1996b) and its extensions are central to the solutions discussed in this paper. Partial evaluation comes in two flavours: offline and online. Section 4.2 basically describes an offline partial evaluator, and Section 4.3 describes an online partial evaluator, though in a limited form. For a more practical use of online partial evaluation in embedding, see Leißa *et al.* (2015). Dybjer and Filinski (2000); Filinski (1999) characterise the relation between partial evaluation and NBE.

5.3 Stand-Alone DSLs

DSLs can also be implemented as a stand-alone language. In theory, for a stand-alone DSL one needs to implement all the required machinery, and they do not integrate easily with other languages. However, there are wide range of tools and frameworks available that automate parts of the implementation process (e.g., see Kats and Visser (2010)). While obviously EBN does not apply to stand-alone language, the original NBE techniques can definitely be used as a way to write normalisers for stand-alone DSLs. In theory, it is even possible to implement tools to automate part of the process.

6. Conclusion

Girard, starts the first chapter of his popular book "Proofs and Types" (Girard *et al.* 1989) by the following paragraph:

Theoretical Computing is not yet a science. Many basic concepts have not been clarified, and current work in the area obeys a kind of "wedding cake" paradigm: for instance language design is reminiscent of Ptolomeic astronomy — forever in need of further corrections. There are, however, some limited topics such as complexity theory and denotational semantics which are relatively free from this criticism.

This paper shows that theoretical Normalisation-By-Evaluation (NBE) techniques, commonly used in denotational semantics, correspond to popular embedding techniques, commonly used in programming practice.

This paper characterises the correspondence, and puts it into practice, by an approach dubbed as Embedding-By-Normalisation (EBN). Then, the paper employs EBN to clarify some of the basic concepts in the practical embedding techniques, concepts such as code extraction (reification) and normalisation. The clarification offered by EBN helps to solve the problem of extracting object code from embedded programs involving sum types, such as conditional expressions, and primitives, such as literals and operations on them.

One final observation of this paper might be that there is science and beauty at the core of the embedding techniques, but it demands rigour and patience to uncover.

Acknowledgements Najd was funded by a Google Europe Fellowship in Programming Technology. Svenningsson was funded by

the Swedish Foundation for Strategic Research under grant RawFP. Lindley and Wadler were funded by EPSRC Grant EP/K034413/1.

References

- A. Abel, K. Aehlig, and P. Dybjer. Normalization by evaluation for martin-löf type theory with one universe. *Electr. Notes Theor. Comput. Sci.*, 173:17–39, 2007.
- T. Altenkirch and T. Uustalu. Normalization by evaluation for lambda². In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*, volume 2998 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2004.
- T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. H. Pitt, D. E. Rydeheard, and P. T. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1995.
- T. Altenkirch, P. Dybjer, M. Hofmann, and P. J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 303–310. IEEE Computer Society, 2001.
- R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.
- E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE*. IEEE, 2010.
- E. Axelsson. A generic abstract syntax model for embedded languages. In P. Thiemann and R. B. Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 323–334. ACM, 2012.
- V. Balat and O. Danvy. Memoization in type-directed partial evaluation. In D. S. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2002.
- V. Balat, R. D. Cosmo, and M. P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 64–76. ACM, 2004.
- U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 203–211, 1991.
- J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.
- M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.
- J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.
- O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.
- O. Danvy. Pragmatics of type-directed partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, volume 1110 of *Lecture Notes in Computer Science*, pages 73–94. Springer, 1996.
- O. Danvy. Type-directed partial evaluation. In H. Boehm and G. L. S. Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 242–257. ACM Press, 1996.
- O. Danvy. Online type-directed partial evaluation. In *Fuji International Symposium on Functional and Logic Programming*, pages 271–295, 1998.
- P. Dybjer and A. Filinski. Normalization and partial evaluation. In *APPSEM*. Springer, 2000.
- P. Dybjer and D. Kuperberg. Formal neighbourhoods, combinatory böhm trees, and untyped normalization by evaluation. *Ann. Pure Appl. Logic*, 163(2):122–131, 2012.
- R. K. Dybvig, S. P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(06):687–730, 2007.
- A. Filinski. A semantic account of type-directed partial evaluation. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, volume 1702 of *Lecture Notes in Computer Science*, pages 378–395. Springer, 1999.
- M. Flatt. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010.
- J. Gibbons and N. Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 339–347. ACM, 2014.
- A. Gill. Domain-specific languages and code synthesis using Haskell. *CACM*, 57(6), 2014.
- J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
- R. J. M. Hughes. A novel representation of lists and its application to the function “reverse”. *Information processing letters*, 22(3):141–144, 1986.
- L. Kats and E. Visser. The spoofax language workbench. In *OOPSLA*. ACM, 2010.
- P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
- R. Leiba, K. Boesche, S. Hack, R. Membarth, and P. Slusallek. Shallow embedding of dsls via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 11–20. ACM, 2015.
- S. Lindley. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. PhD thesis, University of Edinburgh, 2005.
- G. Mainland and G. Morrisett. Nikola: Embedding compiled gpu functions in haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell '10*, pages 67–78, New York, NY, USA, 2010. ACM.
- G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Haskell*. ACM, 2007.
- P. Martin Löf. of. an intuitionistic theory of types: Predicative part. In *Logic Colloquium*, volume 73, page 73, 1973.
- C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.
- E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06*, pages 706–706, New York, NY, USA, 2006. ACM.
- E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .net framework. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706. ACM, 2006.
- S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: quoted domain-specific languages. In M. Erwig and T. Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 25–36. ACM, 2016.

- F. Nielson and H. Nielson. *Two-level functional languages*. Cambridge University Press, 1992.
- U. Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.
- F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI*. ACM, 1988.
- G. D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical computer science*, 1(2):125–159, 1975.
- J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
- T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 2010.
- T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: linguistic reuse for deep embeddings. In *HOSC*. Springer, 2013.
- T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.
- A. Rossberg. 1ml - core and modules united (f-ing first-class modules). In K. Fisher and J. H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 35–47. ACM, 2015.
- T. Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In J. P. Gallagher, C. Consel, and A. M. Berman, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 12-13, 1997*, pages 22–35. ACM, 1997.
- A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*. Springer, 2013.
- J. Svenningsson and E. Axelsson. Combining deep and shallow embedding of domain-specific languages. *Computer Languages, Systems & Structures*, 44, Part B:143 – 165, 2015.
- J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *IFL*. Springer, 2011.
- W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *TCS*, 248(1), 2000.
- P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989.
- P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–55, 1994.
- P. Wadler. Propositions as types. *CACM*, 2015.